Horo - Developer Guide

By: `Team AY1920S1-CS2103T-F12-1` Since: `Sept 2019` Licence: `MIT`

1. Introduction

1.1. About the Application

Horo is a command-line interface scheduling application. It helps the user maintain a to-do list and a calendar, and posts timely reminders on their desktop.

1.2. Purpose

This guide specifies the architecture and software design decisions for Horo, and instructions for building upon the current codebase. This is done in hopes of ensuring extensibility and maintainability of Horo for both current and future developers.

1.3. How to use this Guide

todo

2. Setting up

Refer to the guide here.

3. Design

3.1. Architecture

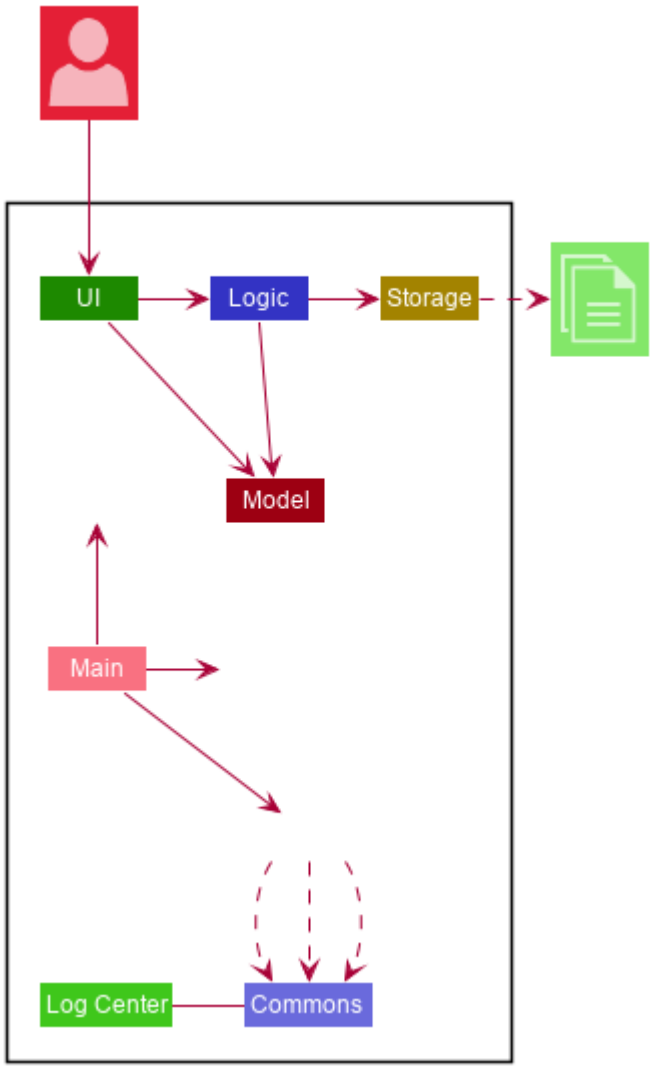


Figure 1. Architecture Diagram

The ***Architecture Diagram*** given above explains the high-level design of the App. Given below is a quick overview of each component.

`Main` has two classes called `Main` and `MainApp`. It is responsible for,

- At app launch: Initializes the components in the correct sequence, and connects them up with each other.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages to the App's log file.

The rest of the App consists of four components.

- **`UI`**: The UI of the App.
- **`Logic`**: The command executor.
- **`Model`**: Holds the data of the App in-memory.
- **`Storage`**: Reads data from, and writes data to the hard disk.

Each of the four components

- Defines its *API* in an `interface` with the same name as the Component.
- Exposes its functionality using a `{Component Name}Manager` class.

For example, the `Logic` component (see the class diagram given below) defines it's API in the `Logic.java` interface and exposes its functionality using the `LogicManager.java` class.

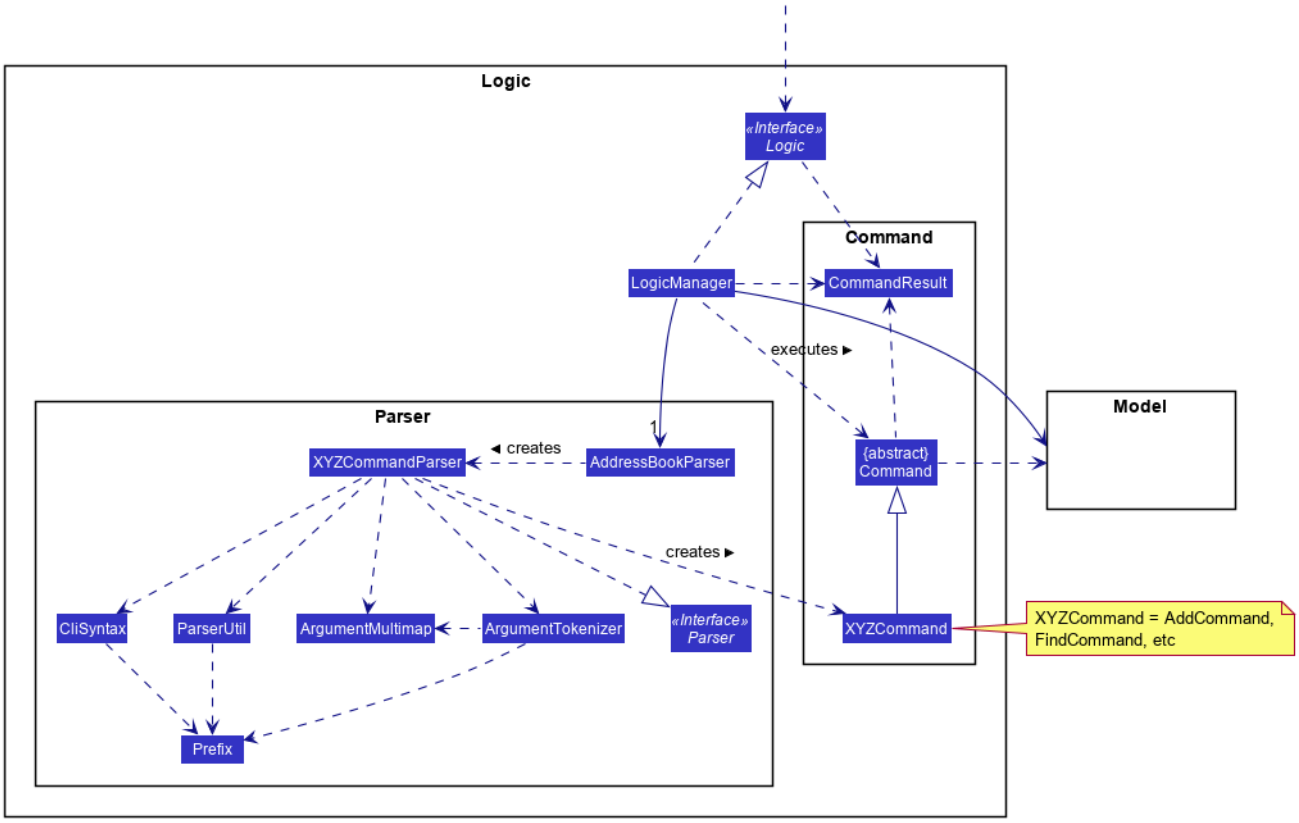


Figure 2. Class Diagram of the Logic Component

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete 1`.

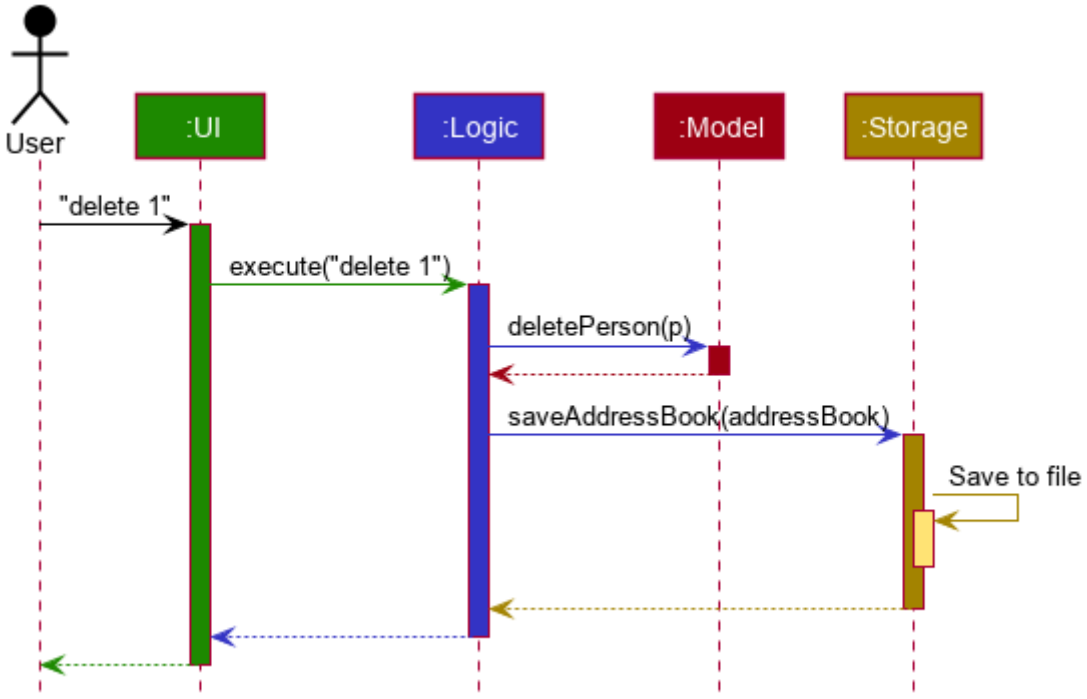


Figure 3. Component interactions for `delete 1` command

The sections below give more details of each component.

3.2. UI component

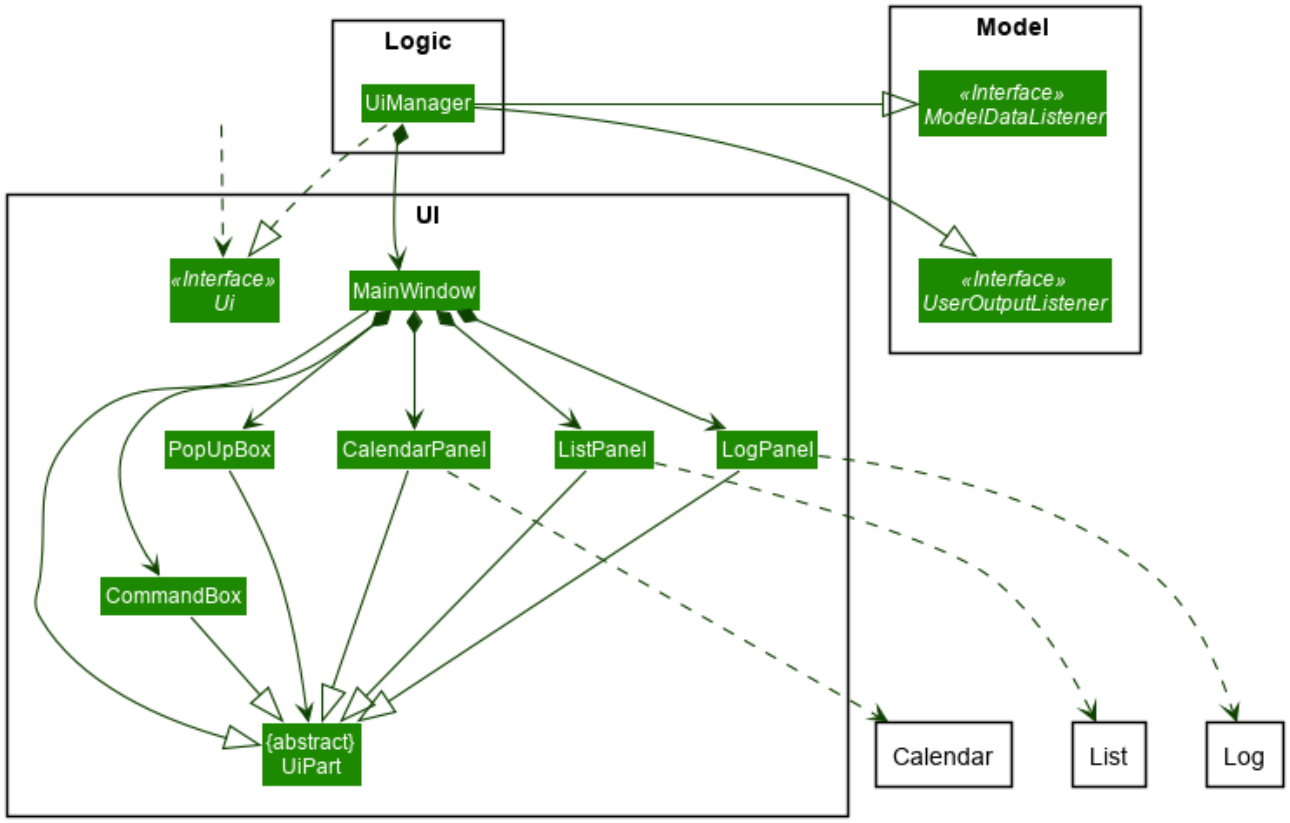


Figure 4. Main structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is contains 3 main view parts - `CalendarPanel`, `ListPanel`, `LogPanel`. It also holds several other UI parts e.g. `PopUpPanel` and `Command Box`. Every one of the UI classes will abstract from the abstract `UiPart` class.

The `UI` component uses JavaFx UI framework, and layout of these UI parts are defined in `.fxml` files which are found in the `src/main/resources/view` folder. One example of the layout would be: `MainWindow`, whose FXML link is specified in `MainWindow.fxml`

The `UI` component does the following:

- Executes user two different types commands using the `Logic` component.
    - One command, when executed, affect the actual Events.
    - The other command is executed to change the view of the UI. There are currently 3 main views in the application: `CalendarPanel`, `ListPanel`, `LogPanel`.
- Listens for any changes in both lists of Tasks and Events using a listener: `ModelDataListener`.

3.2.1. Calendar UI View

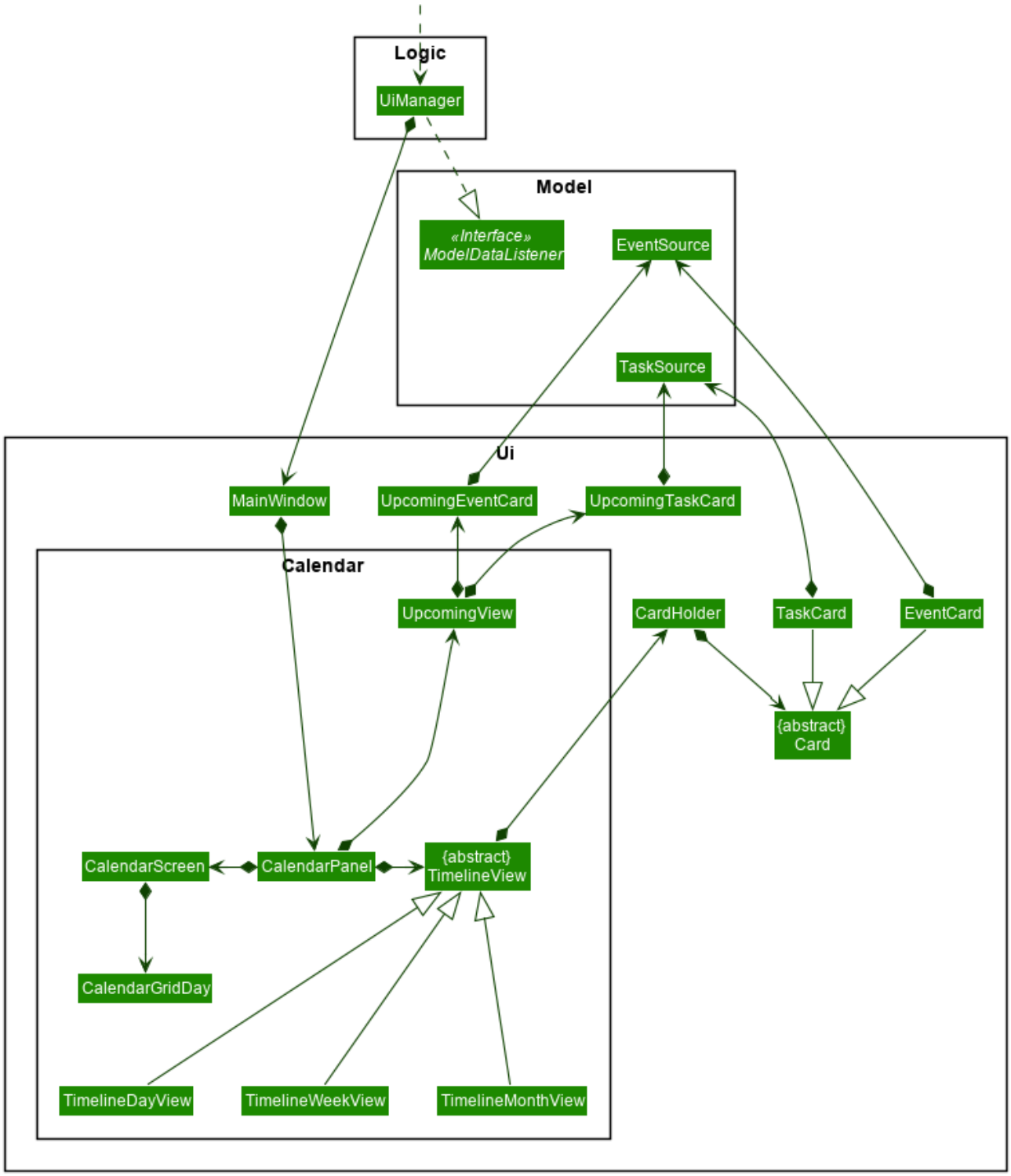


Figure 5. Structure of the Calendar UI View

The Calendar View is made up of the `CalendarPanel`, which holds several different other UI parts linked together to form the overall UI. In the Calendar View, it displays three different UI parts of the Calendar: `CalendarScreen`, `TimelineView` and `UpcomingView`.

`CalendarScreen` is the screen which displays the calendar of a certain month and year to the user. It contains 6 x 7 instance of `CalendarGridDay`, which displays the days of the month.

`TimelineView` is the screen which displays the timeline using 3 different classes which abstract from `TimelineView`.

- `TimelineDayView` displays the timeline of a particular day in a certain month and year.
- `TimelineWeekView` displays the timeline of a particular week. The week is according to the `CalendarScreen`, where each row represents a week of a month.
- `TimelineMonthView` displays the timeline of a particular month in a certain year.

Each of these timeline will hold up to a certain amount of `CardHolder` depending on the type of `TimelineView`. Each of these `CardHolder` will then hold an amount of `Card` for displaying the event name and date. The details of `Card` will be explained in the one of the next few sections.

`UpcomingView` represents a miniature list of Events and Tasks that has a start date or due date in the same month as the user's system current month, but not before the date as the user's date. This list will then hold up to a certain amount of `UpcomingEventCard` and `UpcomingTaskCard` which will be explained together with `Card` as well.

3.2.2. List UI View

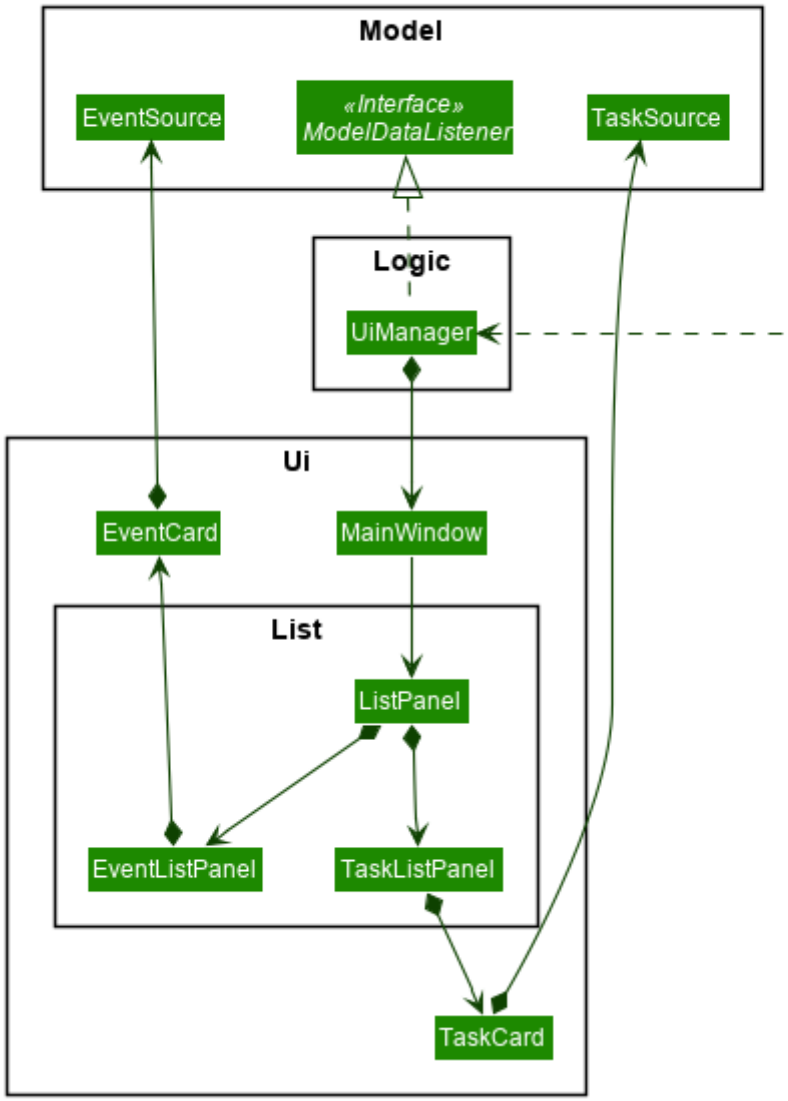


Figure 6. Structure of the List UI View

The List View is made up of the `ListPanel` which contains two lists views, `EventListView` and `TaskListView`

- `EventListView` displays the list of Events containing every piece of information of the Events.
- `TaskListView` displays the list of Task, containing every piece of information of each Task.

Similar to `TimelineView`, `EventListView` and `TaskListView` will contain a list of `Card` to display the information.

3.2.3. Log UI View

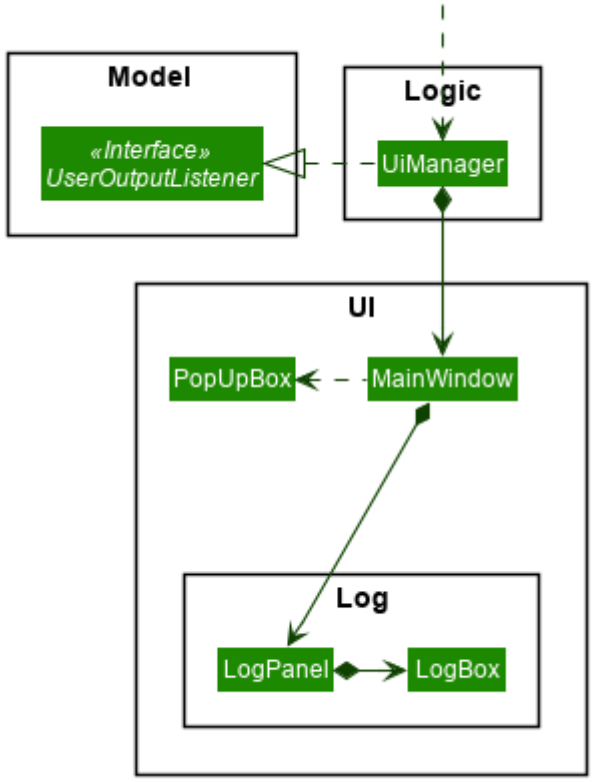


Figure 7. Structure of the Log UI View

The Log View is made up of the `LogPanel` which contains the list of `LogBox`.

`LogBox` displays literal information back to the user when it is called by `MainWindow` when it listens for a command.

`PopUpBox` is rather similar to LogBox. It holds up to the same amount of information, as much until the size of the application window, and collapses the rest. It represents the pop up that animates and displays for a few seconds to the user about the given command.

3.2.4. Card UI

Firstly, there are two types of ways to display information to the user regarding a Event or Task.

- For Events, it is `EventCard`, which is abstracted from the `Card` abstract class, followed by `UpcomingEventCard`
- For Tasks, it is `TaskCard`, which is abstracted from the `Card` abstract class, followed by

`UpcomingTaskCard`.

An `EventCard` may display the following information:

1. Event Description
2. Event Start Date
3. Event End Date (Optional)
4. Event Reminder Date (Optional)
5. Event Tags (Optional)
6. Event Index (For deleting or editing)

An `TaskCard` may display the following information:

1. Task Description
2. Task Due Date (Optional)
3. Task Reminder Date (Optional)
4. Task Tags (Optional)
5. Task Index (For deleting or editing)

As for `UpcomingEventCard` and `UpcomingTaskCard`, they only hold the Description of the Event or Task.

3.3. Logic component

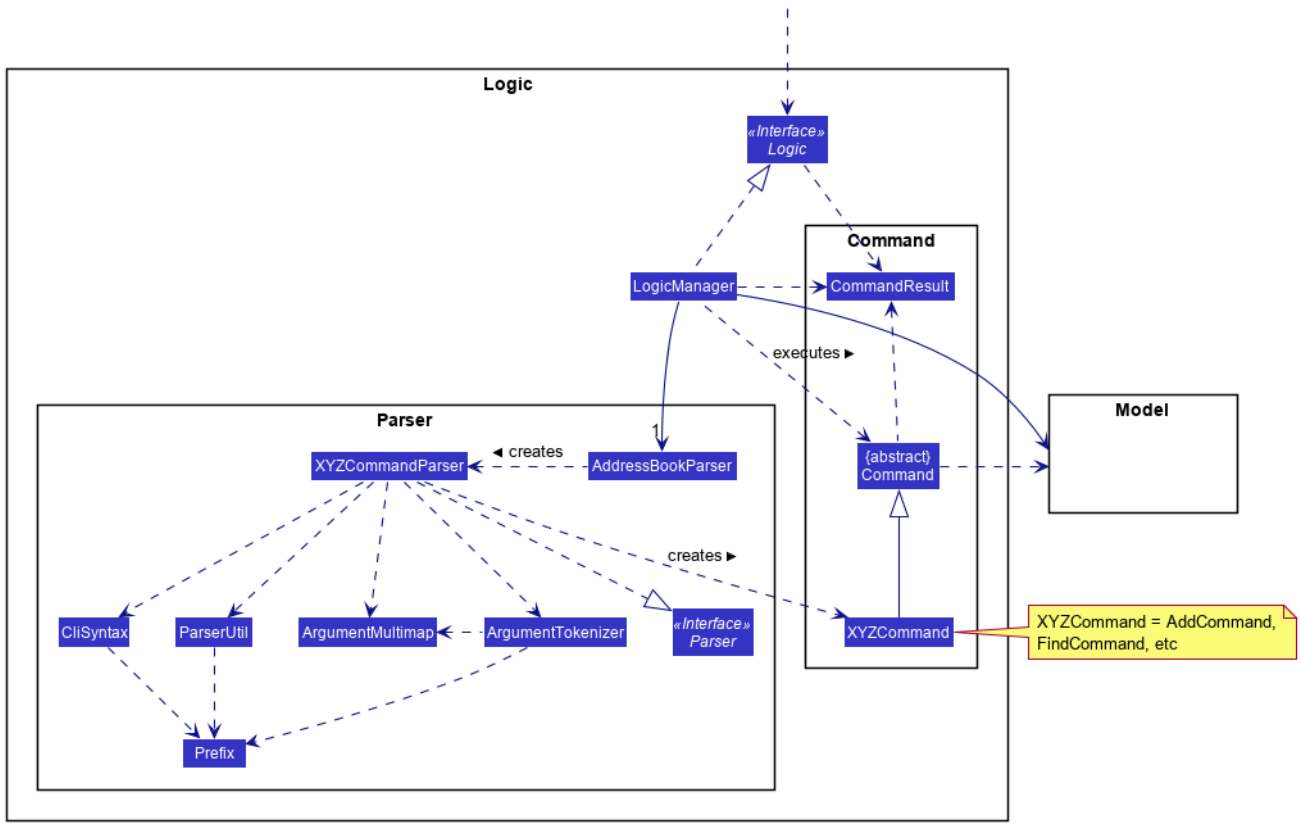


Figure 8. Structure of the Logic Component

API : `Logic.java`

1. `Logic` uses the `AddressBookParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a person).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

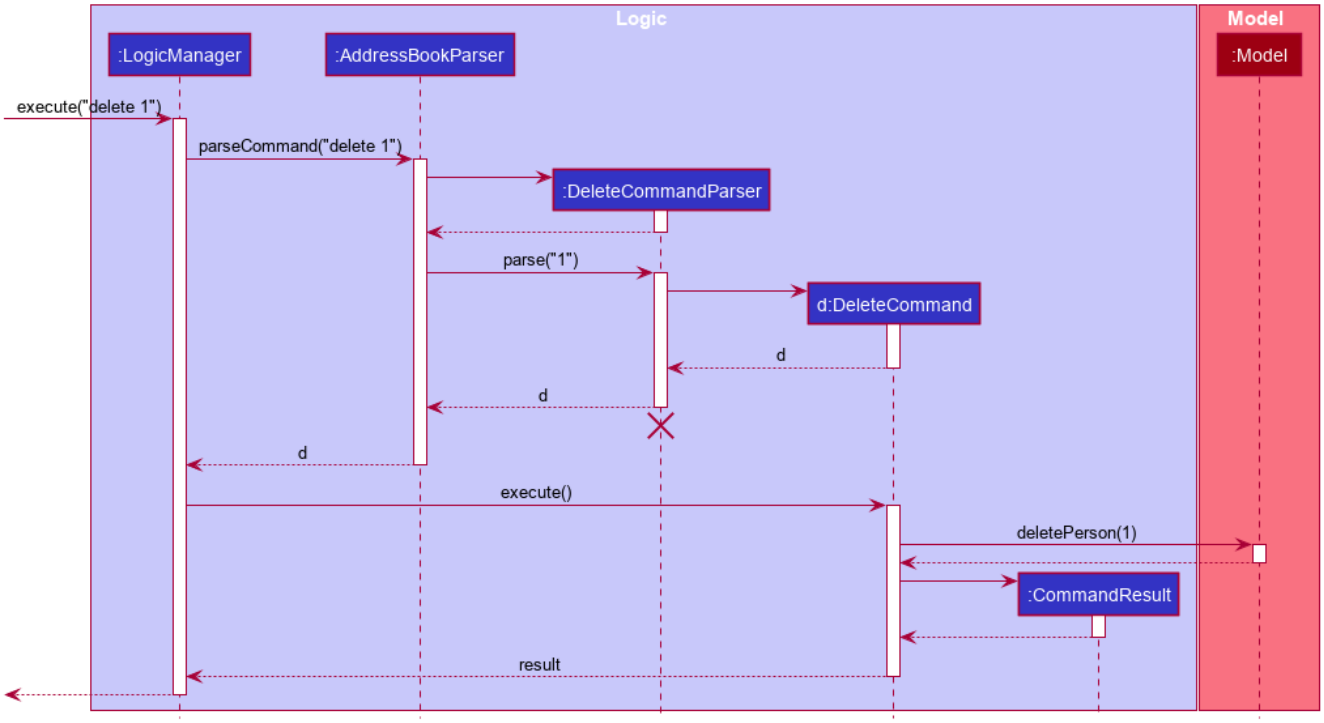


Figure 9. Interactions Inside the Logic Component for the `delete 1` Command

NOTE	The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

3.4. Model component

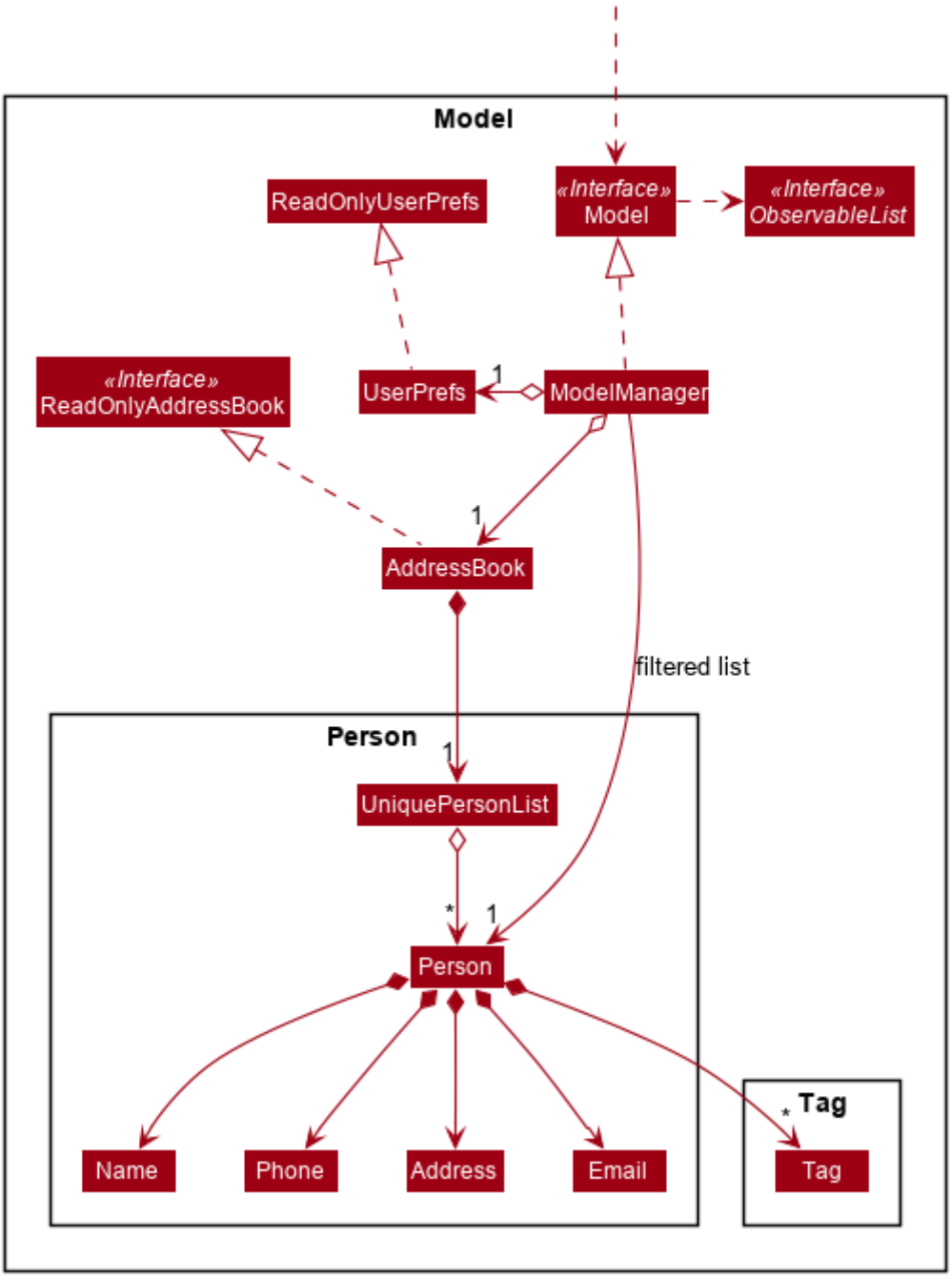


Figure 10. Structure of the Model Component

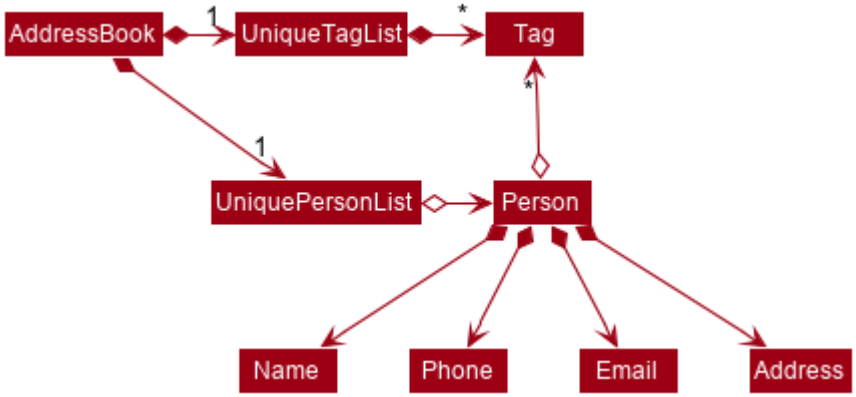
API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user's preferences.
- stores the Address Book data.
- exposes an unmodifiable `ObservableList<Person>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

NOTE

As a more OOP model, we can store a `Tag` list in `Address Book`, which `Person` can reference. This would allow `Address Book` to only require one `Tag` object per unique `Tag`, instead of each `Person` needing their own `Tag` object. An example of how such a model may look like is given below.



+

3.5. Storage component

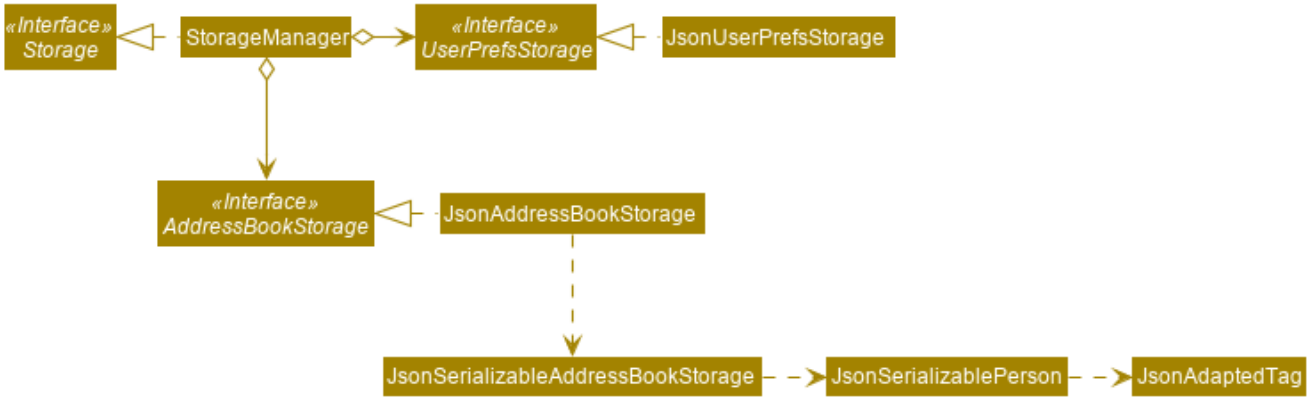


Figure 11. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the Address Book data in json format and read it back.

3.6. Common classes

Classes used by multiple components are in the `seedu.addressbook.commons` package.

4. Implementation

This section describes some noteworthy details on how certain features are implemented.

4.1. Undo/Redo feature

4.1.1. Implementation Details

The undo/redo mechanism is facilitated by `UndoRedoManager`, which contains `undoStateList` - a history of `ModelLists`. Each `ModelList` object contains two lists: one to store `EventSources` and the other to store `TaskSources`, together representing the state of all event and task data at that point in time. `UndoRedoManager` also contains a `undoIndex`, which keeps track of the index of the `ModelList` being used presently.

A key point to take note of is that `ModelManager` contains an `eventList` (a list of `EventSources`) and a `taskList` (a list of `TaskSources`), both of which cannot be reinitialized. This is because these two lists are directly in sync with the GUI; changes to these specific list instances are reflected as changes to the GUI, but changes to other copies of `EventSource` or `TaskSource` lists will not affect the GUI. Hence, the history of `ModelLists` held by `UndoRedoManager` stores deep-copies of `EventSource` and `TaskSource` lists. Should there be a need to revert back to a past or future state (if undo or redo is called), these lists will retrieve their data from the appropriate copy of `ModelList` in the list of duplicates.

`UndoRedoManager` also implements the following operations:

- `UndoRedoManager#commit(ModelList state)` — Adds the new state (which contains a deep-copied version of `TaskSource` and `EventSource` lists) to the `undoStateList`
- `UndoRedoManager#undo()` — Restore `eventList` and `taskList` in `ModelManager` to their previous versions from the appropriate duplicate in `undoStateList` via an `ModelListListener`
- `UndoRedoManager#redo()` — Restore `eventList` and `taskList` in `ModelManager` to their future versions from the appropriate duplicate in `undoStateList` via an `ModelListListener`
- `UndoRedoManager#clearFutureHistory()` -- Delete all `ModelList` states that occur in `undoStateList` after the index given by the `undoIndex`

The `UndoCommand` and `RedoCommand` will interact directly with `UndoRedoManager` while other state-changing commands (such as adding or deleting tasks) will interact only with `ModelManager`.

There are two key **Listener** interfaces that help us achieve the desired undo-redo functionality:

- `ModelListListener`
- `ModelResetListener`

These listener interfaces each contain a single method, `ModelListListener` contains `onModelListChange(ModelList list)` and `ModelResetListener` contains `onModelReset(ModelList state, Object caller)`.

The `UndoRedoManager` implements the `ModelListListener` interface's method `onModelListChange(ModelList list)` to "listen" for any changes to `ModelManager`'s lists (`eventList` or `taskList`). (e.g. when an event or task is added or deleted) If such a change exists, it will be handled by first instantiating a `ModelList` with a deep-copied version of the `taskList` and the modified `eventList`, calling `UndoRedoManager#clearFutureHistory()`, and calling `UndoRedoManager#commit(ModelList state)` to commit the state.

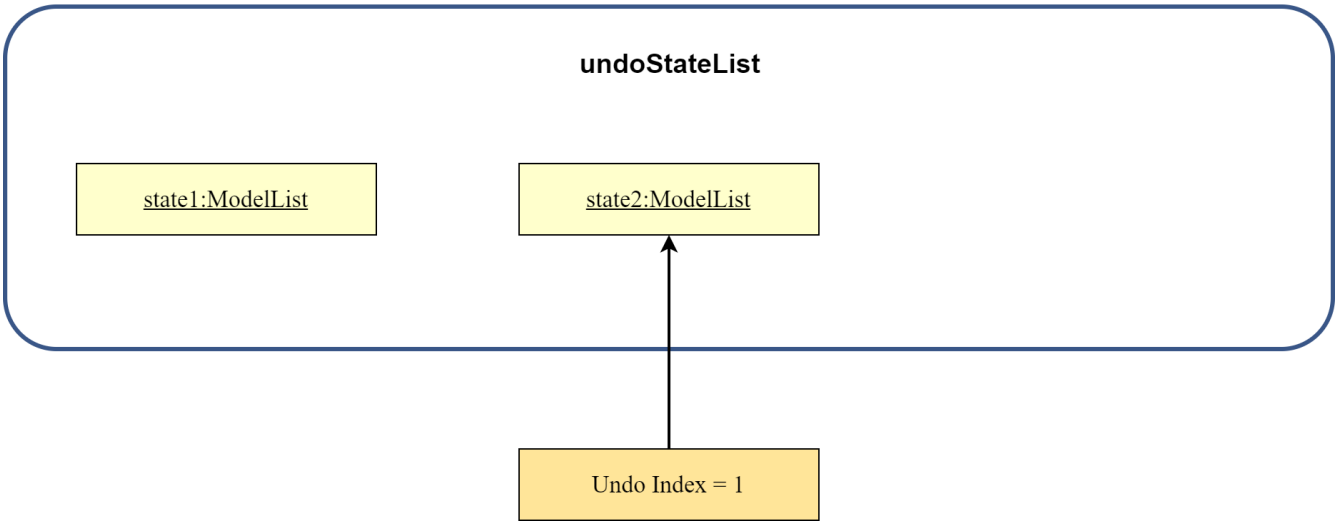
The `ModelManager` implements the `ModelResetListener` interface's method `onModelReset(ModelList state, Object caller)` to "listen" for any undo or redo being executed. This will be handled by resetting `ModelManager`'s `eventList` and `taskList` data to match the data of the `ModelList` with index `undoIndex` in `undoStateList`.

Given below is an example usage scenario and how the undo/redo mechanism behaves at each step.

Step 1. The user runs the program for the first time. The `UndoRedoManager` will be initialized with the initial `undoStateList`. A `ModelList` will be added to `undoStateList` and the `undoIndex` will point to that single `ModelList` in the list.

Step 2. The user executes `add_event "Suntec City Computer Fair" --at "17/11/2019 12:00"`. The event will be added to `ModelManager`'s `eventList`. Then, `UndoRedoManager#onModelListChange(ModelList list)` will be called (as there has been a change to the `eventList`), deep-copying the modified `eventList` and `taskList` and instantiating a new state `ModelList` with these copies. All future states beyond the `undoIndex` will be cleared as they are no longer useful. In this particular case, there are no future states to be cleared. Finally, the new `ModelList` state will be committed; added to `undoStateList`. The `undoIndex` is incremented by one to contain the index of the newly inserted model list state.

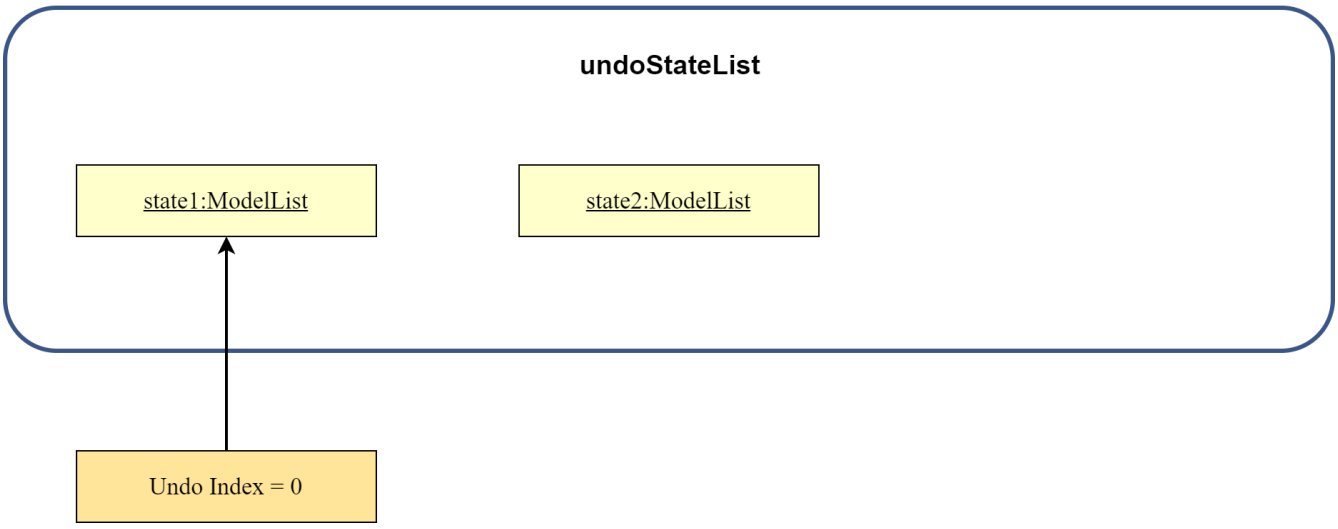
After command "add_event..."



NOTE If a command fails its execution, it will not result in any change to `ModelManager#eventList` or `ModelManager#taskList`. Hence, there is no change to trigger the listener methods and thus no `ModelList` will be saved to `undoStateList`.

Step 3. Suppose the user decides that adding the task was a mistake. He/she then executes the undo command to rectify the error. The undo command will decrement the `undoIndex` by one to contain the index of the previous undo redo state, thereafter triggering the `ModelManager#onModelReset` method. This method updates `ModelManager`'s `eventList` and `taskList` data to match the data of the `ModelList` with index `undoIndex` in `undoStateList`.

After command "undo"



NOTE If the `undoIndex` is 0, pointing to the initial model list state, then there are no previous model list states to restore. The undo command uses `UndoRedoManager#canUndo()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the undo.

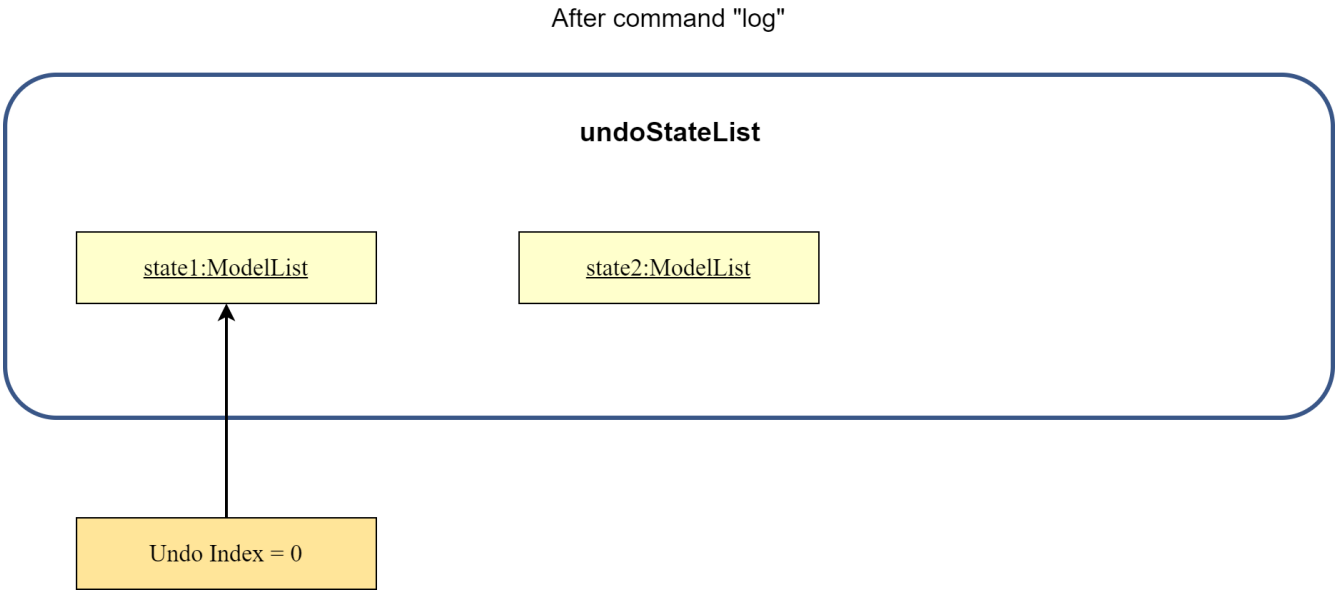
The following sequence diagram shows how the undo operation works:

NOTE The lifeline for `UndoCommand` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

The redo command does the opposite — it calls `UndoRedoManager#redo()`, which increments the `undoIndex` by one to contain the index of the previously undone state. The `ModelResetListener` then causes `ModelManager#eventList` and `ModelManager#taskList` to reset their data to this state's list data.

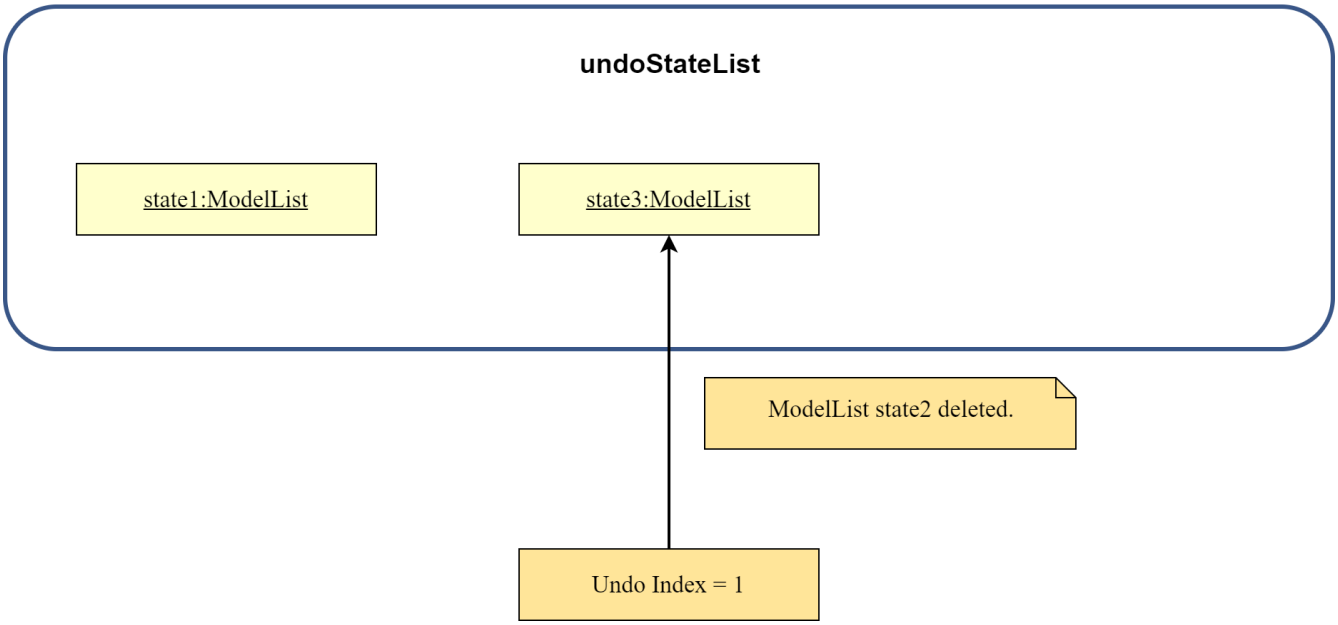
NOTE If the `undoIndex` is at index `undoStateList.size() - 1`, pointing to the latest model list state, then there are no undone model list states to restore. The `redo` command uses `UndoRedoManager#canRedo()` to check if this is the case. If so, it will return an error to the user rather than attempting to perform the redo.

Step 4. The user decides to execute the command `log`. Non-state-changing commands such as `log` do not manipulate task and event data. Since no changes to `taskList` or `eventList` have been made, the listener methods will not be triggered and no `ModelList` will be saved to `undoStateList`. Thus, the `undoStateList` remains unchanged.

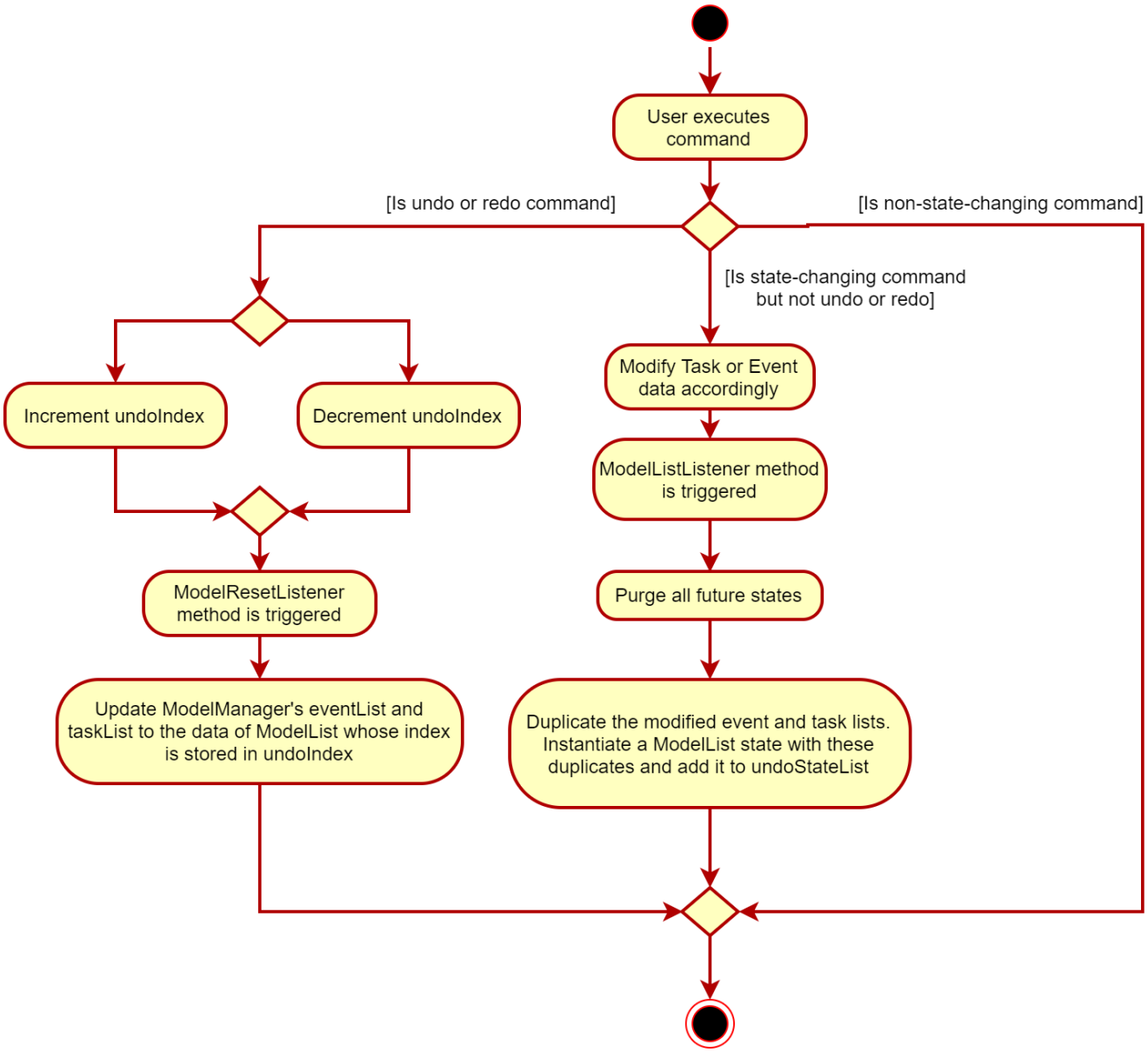


Step 5. The user executes `delete_event 1`, removing the event from `ModelManager`'s `eventList`. `UndoRedoManager#onModelListChange(ModelList list)` will be called (as there has been a change to the `eventList`), purging all future states beyond the `undoIndex` as they are no longer useful. The modified `eventList` and `taskList` will be deep-copied and a new `ModelList` containing the deep-copies will also be added to `undoStateList`. The `undoIndex` is incremented by one to contain the index of the newly inserted model list state.

After command "delete_event 1"



The following activity diagram summarizes what happens when a user executes a new command:



4.1.2. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves `EventSource` and `TaskSource` data every time a change has been made.
  - Pros: Easy to understand and implement.
  - Cons: Performance issues may arise due to the relatively larger memory usage required.
- **Alternative 2:** Individual command knows how to undo/redo by itself; inverse functions have to be implemented (if I undo the deletion of a person, it would be equivalent to adding him back to the list)
  - Pros: Uses less memory as we only need to keep track of what commands have been executed and their parameters, as opposed to storing all task and event data between every change.
  - Cons: Every command will have to be implemented twice, since their inverse operations will all be different. This is compounded by the fact that we have to ensure the correctness of

every inverse operation individually as well.

4.2. Notification System

4.2.1. Class Architecture

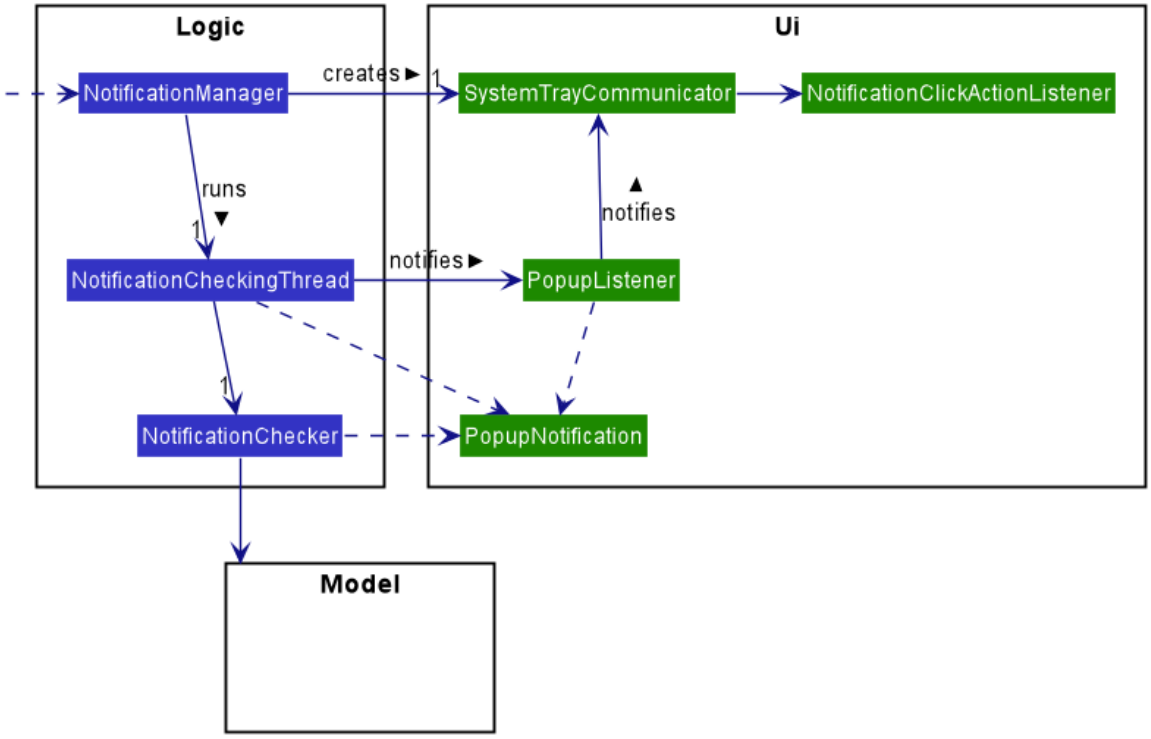


Figure 12. Class diagram for Notification System

The Notification System is facilitated by the `NotificationManager`, which is found in the Logic component. Other constituent classes of the Notification System can be found in the Logic and UI components, depending on their functionality. These classes and their functionalities are listed below:

Logic Classes

Logic classes are responsible for deciding if a notification should be posted. As with other components, their functionality is accessed through the `NotificationManager` class. The `NotificationManager` class maintains a reference to a `NotificationCheckingThread` as well as a `SystemTrayCommunicator`.

The logic classes of the Notification System can be found under the `notification` package under the `Logic` component.

- The `NotificationCheckingThread` is a daemon thread that runs in parallel with the main application. It checks for new notifications to post every minute.
- The `NotificationChecker` is responsible for checking `Model` for any notifications that need to be posted.

UI Classes

UI classes are responsible for displaying notifications to the user.

The UI classes of the Notification System can be found under the `systemtray` package under the `ui` component.

- The `PopupListener` class is the main channel of communication between the logic and UI classes. When a notification needs to be posted, it will relay the information from the logic to UI classes.
- The `SystemTrayCommunicator` handles posting notifications and displaying the app's icon on the System Tray. It listens to the `NotificationCheckingThread` through a `PopupListener`.
- The `PopupNotification` class carries the information that will be posted to a popup notification.
- The `NotificationClickActionListener` is called when the user clicks on a popup notification.

4.2.2. Class Behaviour

As with other Manager classes, an instance of the `NotificationManager` is created upon the starting of MainApp. The NotificationManager proceeds to initialize and run a `NotificationCheckingThread`, as well as a `SystemTrayCommunicator`. Upon being started, the `NotificationCheckingThread` will enter a `notificationCheckingLoop` by calling its method of the same name.

To give a better explanation of how the `NotificationCheckingThread` works, a single run of its loop is illustrated below:

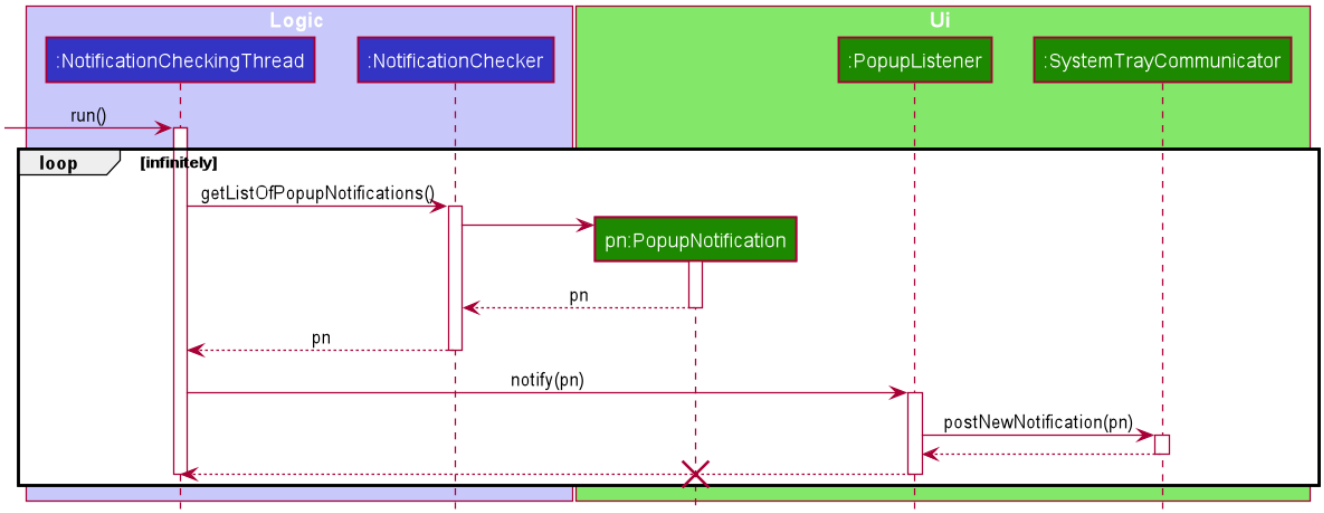


Figure 13. Sequence diagram for NotificationCheckingThread's main loop

Step 1. The `NotificationCheckingThread` calls the `NotificationChecker` to generate instances of `PopupNotification` through a call to `NotificationChecker#getListOfPopupNotifications()`

Step 2. For each `PopupNotification` generated by the NotificationChecker, a call to `PopupListener#notify()` is made.

Step 3. This prompts the `SystemTrayCommunicator` to post a new notification.

Step 4. The `NotificationCheckingThread` sleeps until the start of the next minute, found by the method `NotificationCheckingThread#findMillisecondsToNextMinute()`.

4.2.3. Design Considerations

Aspect: How the Notification system should run

- **Alternative 1 (current choice):** Running the Notification system as a separate thread in the same application
  - Pros: Easier to implement and test.
  - Cons: The user would have to leave the application on if they always wanted to be notified.
- **Alternative 2:** Running the Notification system as a background application
  - Pros: This would allow notifications to be posted to the user's desktop even if the Horo main app were not open.
  - Cons: This would require the creation of a separate application that the user would have to install on their computer. Because different Java applications are ran in different instances of Java Virtual Machines, this could vastly complicate implementation as the Notification System and the rest Horo would be unable to interact directly.

Alternative 1 was eventually chosen as it was simpler to implement and test, and remain within the initial scope of Horo's development. The application can be potentially changed to use Alternative 2 in the future.

4.3. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See [Implementation-Configuration])
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application
- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

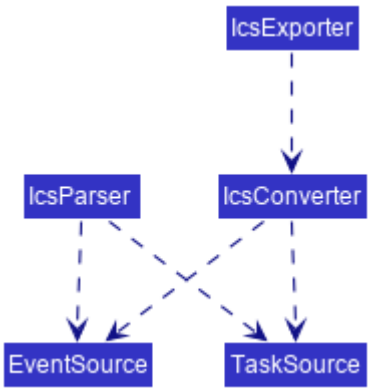
4.4. Ics Component

API : `IcsParser.java`

The ICS component is made up of 2 main sub-components: ICS file parser, and ICS file exporter.

The file parser makes use of a custom parser that converts files with the `.ics` file extension to `EventSource` and `TaskSource` objects in Horo.

Here is an overview of how the ICS component looks like:



The file exporter uses the `IcsConverter` class to convert `EventSource` and `TaskSource` objects stored in the `ModelManager` singleton object into their ICS String representations. Check out the iCalendar Wiki Page for more information on the specifications.

- Can export Horo's save data as a file The `ICS` Component,with a `.ics` extension.
- Can import other Horo's save data from a `.ics` file.

4.4.1. Design Considerations

Aspect: Handling of Horo TaskSource and EventSource conversion to ICS Strings

- **Alternative 1 (Current Choice):** Use of a separate class `IcsConverter` to convert `TaskSource` and `EventSource` objects their ICS string representations.
  - Pros: Adherence to Single Responsibility Principle, decouples `IcsExporter` from the `TaskSource` and `EventSource` classes, and keeps code reusable and scalable.
  - Cons: Not consistent with Object-Oriented Programming structure.
- **Alternative 2:** Create a common `IcsConvertible` Interface for `TaskSource` and `EventSource` to implement a `toIcsString()` function.
  - Pros: Adheres to Object-Oriented Programming structure.
  - Cons: Hard to reuse functions and modify code.

Alternative 1 was chosen eventually, as I felt that it is more important to adhere to the Single Responsibility Principle and keep all code relevant to converting objects to ICS Strings in the same class.

This further makes it easier for future debugging, and makes adding new exportable objects a lot easier as there are common functions that can be used.

4.5. UI Component

4.5.1. Implementation during change in Events and Tasks

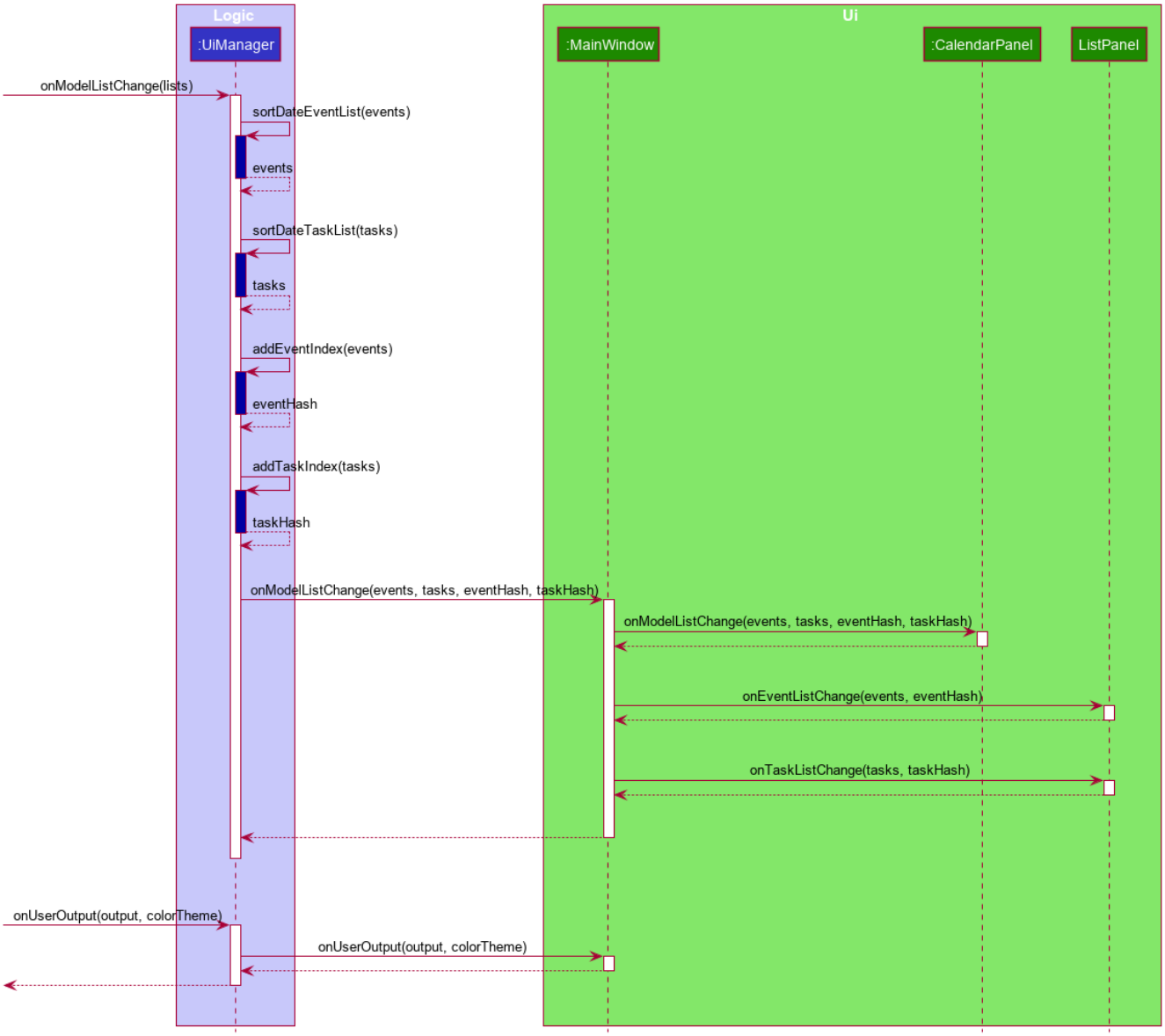


Figure 14. A general Sequence Diagram during a change in the Event and Task Lists model.

The UI system is managed by the `UiManager`, that is found in `Logic` component and is responsible for any change in the models and hence updating the necessary UI portions. The UiManager then holds a single instance of the `MainWindow`, which is the bsae of the UI and holds every other smaller components and views of the rest of the UI. Here is the sequence of a change in Events and Tasks for the UI.

Step 1. `UiManager` will be called using `onModelListChange(lists)` method. This will in turn, take in the `ModelLists`, split them into the `events` and `tasks`, and sort them. Afterwards, two HashMaps, `eventHash` and `taskHash` is created to deal with the indexing of the Ui later on.

Step 2. `MainWindow` will be called by `UiManager` using `onModelListChange(events, tasks, eventHash, taskHash)`, which will in turn proceed to call the methods that will update the different views represented by:

- `CalendarPanel` - `onModelListChange(events, tasks, eventHash, taskHash)`
- `ListPanel` - `onEventListChange(events, eventHash)` and `onTaskListChange(tasks, taskHash)`

Step 3. `UiManager` will also be called using `onUserOutput(output, colorTheme)`, which will in turn call `onUserOutput(output, colorTheme)` for MainWindow.

As for these 3 main panels, each of them will be explained further below

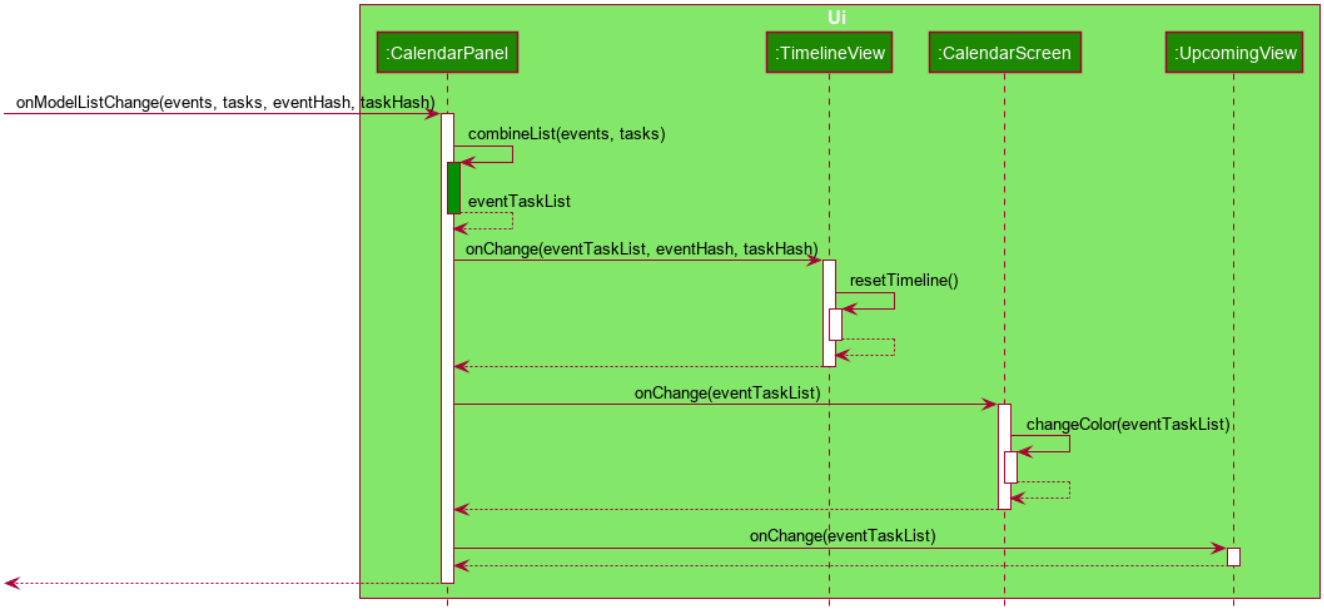


Figure 15. Sequence Diagram for `CalendarPanel`

Step 2.1. `CalendarPanel` will be called by `onModelListChange(events, tasks, eventHash, taskHash)`, and will proceed to zip the two list into a single list for sorting purpose.

Step 2.2. Afterwards, it will call `onChange` for the 3 smaller components:

- `TimelineView` - When called, it will reset the current timeline using `resetTimeline()`
- `CalendarScreen` - When called, it will change the calendar to the given date, as well as calling `changeColor(eventTaskList)` to change the color of a day in the calendar.
- `UpcomingView` - When called, it will simply reset the view to input the correct events and tasks.

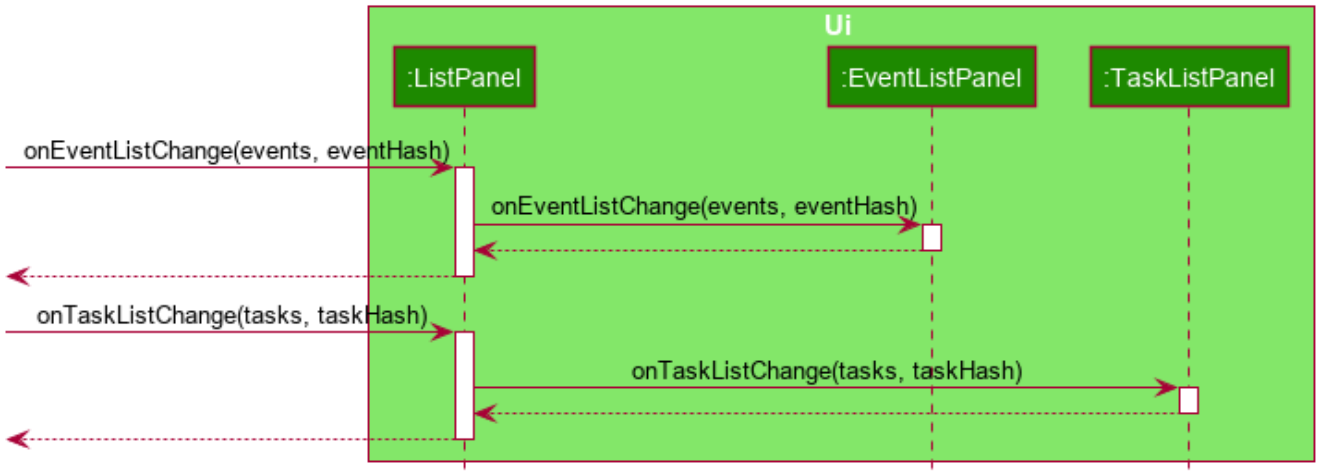


Figure 16. Sequence Diagram for `ListPanel`

Step 2.3. `ListPanel` will be called with `onEventListChange(events, eventHash)` first. It will proceed to call `EventListPanel` to change the list according to the given list of events.

Step 2.4. `ListPanel` will also be called with `onTaskListChange(tasks, taskHash)`. It will afterwards call `TaskListPanel` to change the list accordingly as well.

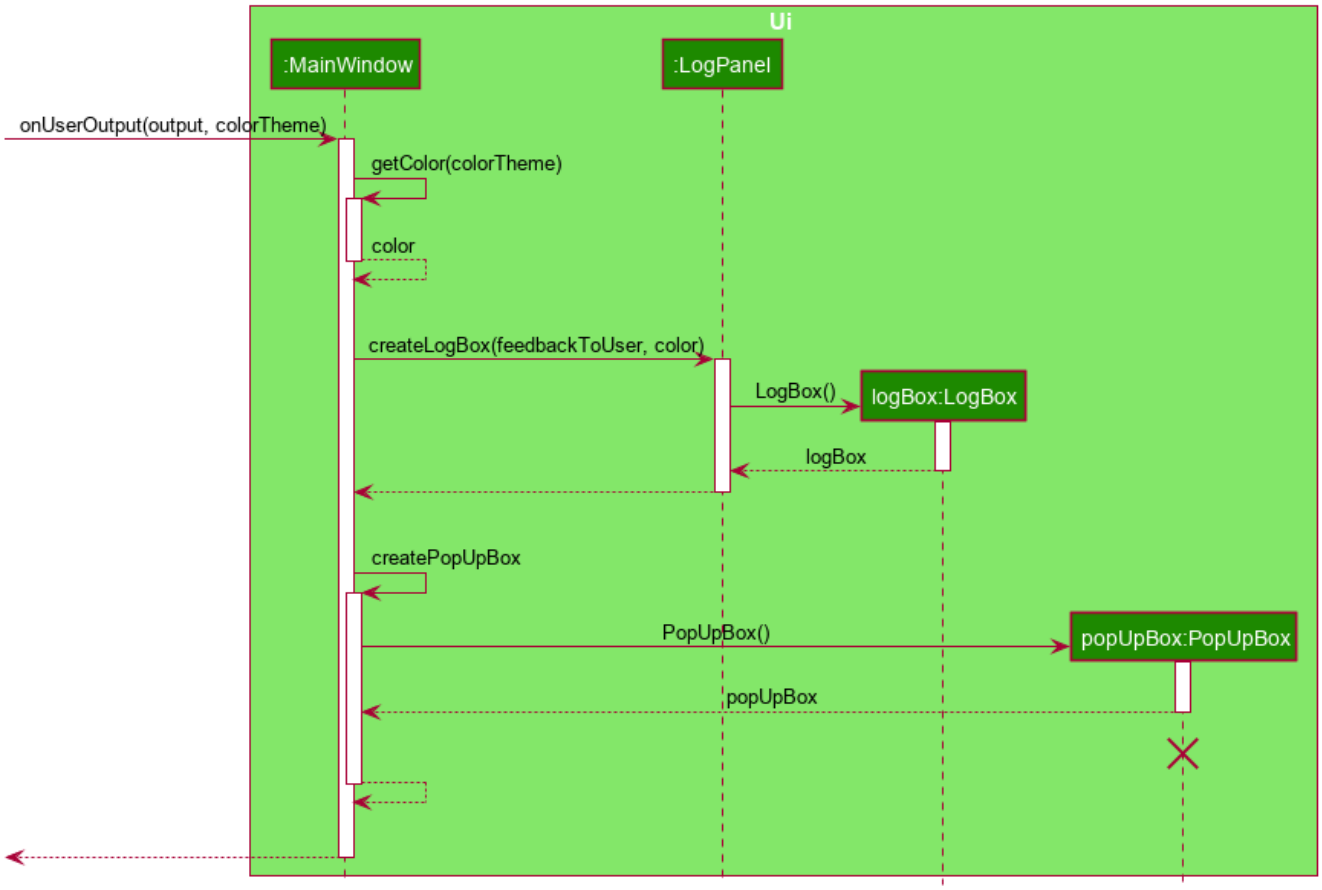


Figure 17. Sequence Diagram for `LogPanel`

Step 3.1. When `MainWindow` gets called using `onUserOutput(output, colorTheme)`, it will proceed to get the actual color scheme, and create 2 different boxes to display the output.

Step 3.2. It will call `LogPanel` to create a `LogBox` using `createLogBox(feedbackToUser, color)` to display the output to the user in `LogPanel`

Step 3.3. Next, it create `PopUpBox` and display it temporarily on any of the panels, and proceed to unused afterwards.

4.5.2. Implementation when changing the date of timeline

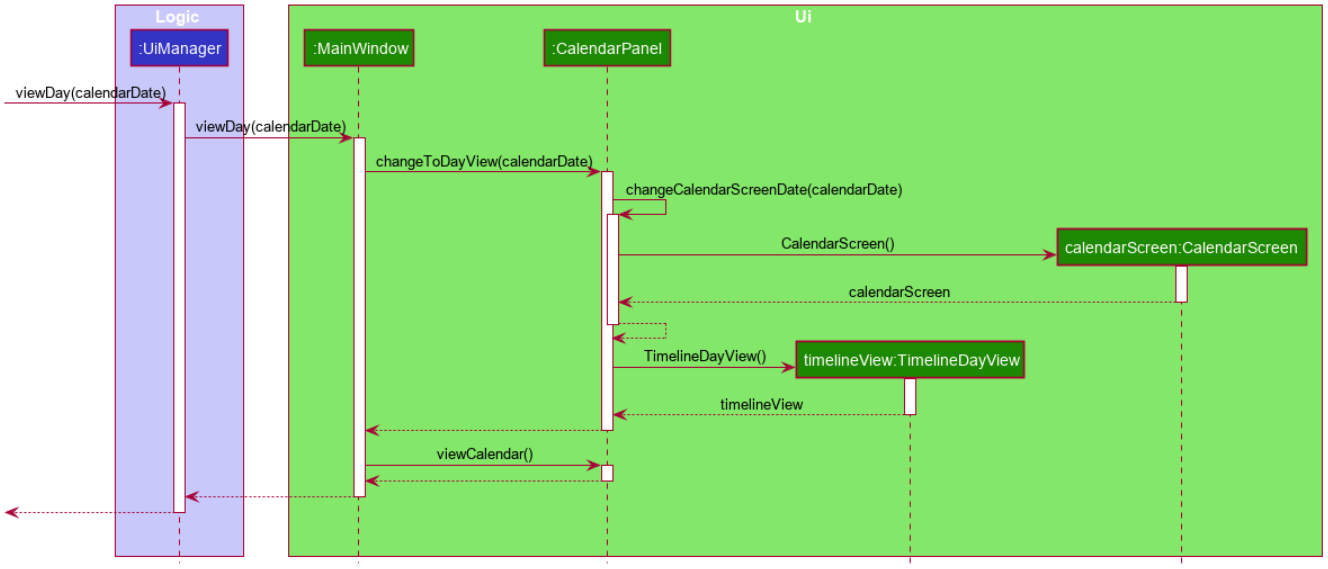


Figure 18. Sequence Diagram for changing the timeline date

Here is an example of the sequence for the UI when `DayViewCommand` is executed to change the date of the timeline.

Step 1. When the command is executed, it will proceed to call `UiManager` through `viewDay(calendarDate)`, which in turn will call `MainWindow` and subsequently `CalendarPanel`.

Step 2. `CalendarPanel` will proceed to execute `changeCalendarScreenDate(calendarDate)`, which will create an instance of `CalendarScreen` to display the calendar.

Step 3. Afterwards, a new instance of `TimelineDayView` will be created to display the timeline.

Step 4. Lastly, `MainWindow` will call `viewCalendar` which will be explained in the next section, so as to allow `CalendarPanel` to be visible while the other panels are not.

4.5.3. Implementation when changing views

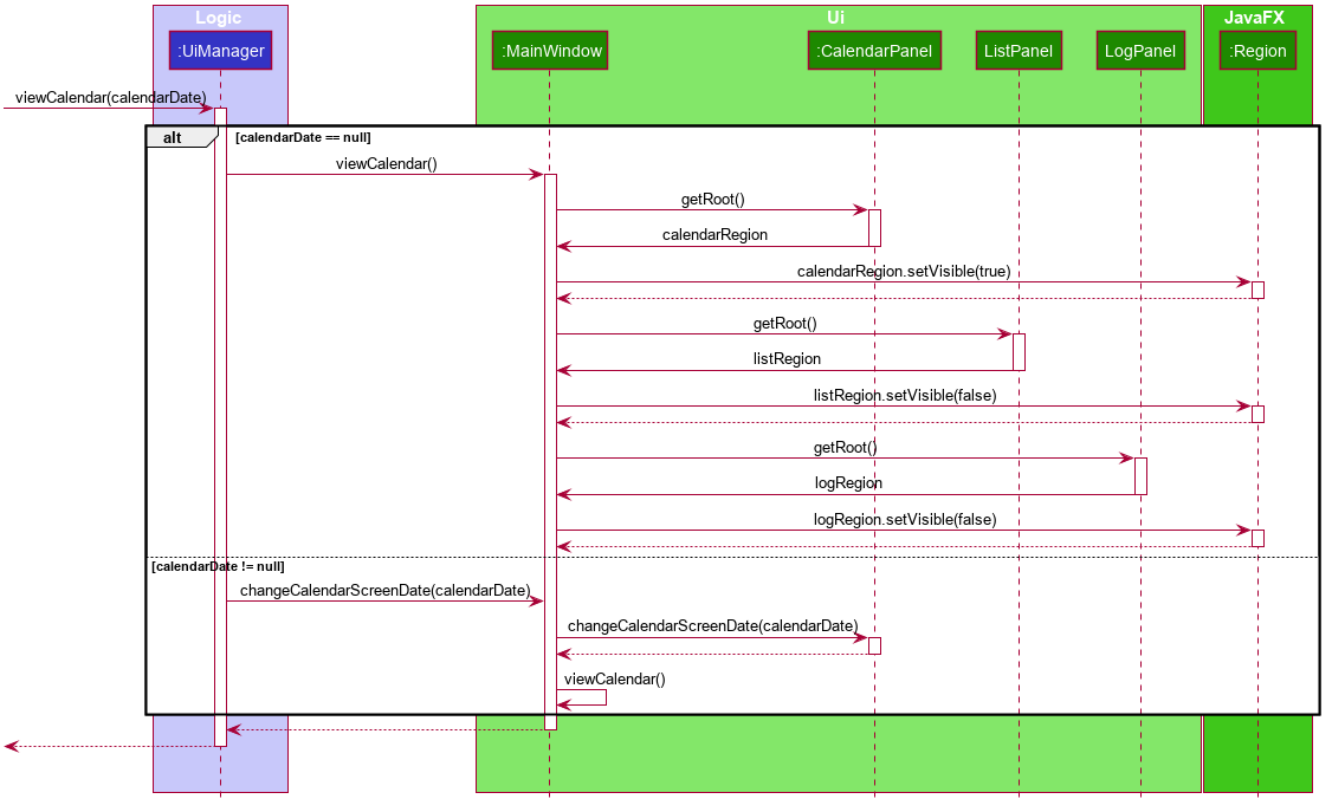


Figure 19. Sequence Diagram for changing to Calendar View

Here is an example of the sequence for the UI when `CalendarViewCommand` is executed.

Step 1. When command is executed, it will proceed to call `UiManager` through `viewCalendar(calendarDate)`, which will proceed to check if the giving date is null, or a date. The validity check is previously check in the parser.

Step 2. If calendarDate is null, the `UiManager` will simply call `MainWindow` to switch the view with the method `viewCalendar()`.

Step 3. `MainWindow` will obtain the `Region` of the 3 panels: `CalendarPanel`, `ListPanel` and `LogPanel`, and proceed to set only `CalendarPanel` to be visible.

Step 4. If calendarDate is not null, `UiManager` will then call `MainWindow` using `changeCalendarScreenDate(calendarDate)`, to change the `CalendarScreen` to the given date.

Step 5. Afterwards, it will proceed to repeat step 3 again, which is simply calling `viewCalendar()` again.

Since the sequence for `CalendarViewCommand` is roughly similar, or in fact, more complicated than `ListViewCommand` and `LogViewCommand`, it will not be further explained.

4.5.4. Design Considerations

The design considerations are more towards how the UI would have look like, as well as how the architecture of the code would have change if depending on the arrangement of the UI.

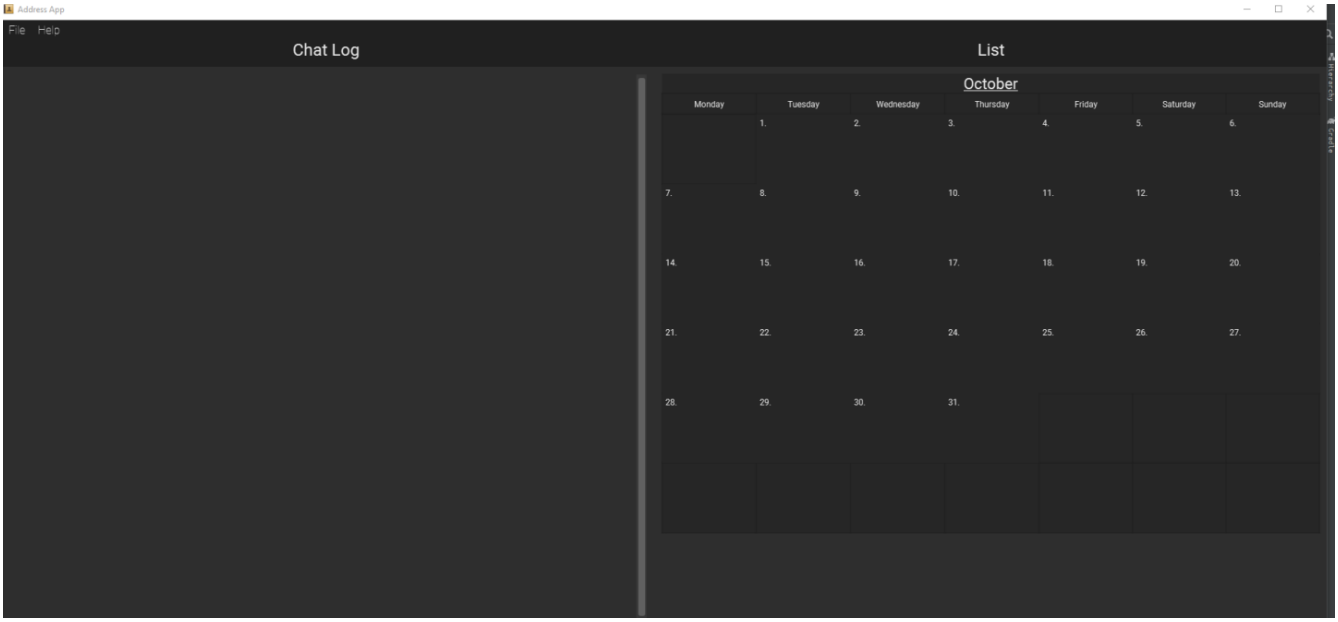


Figure 20. Old design of the UI

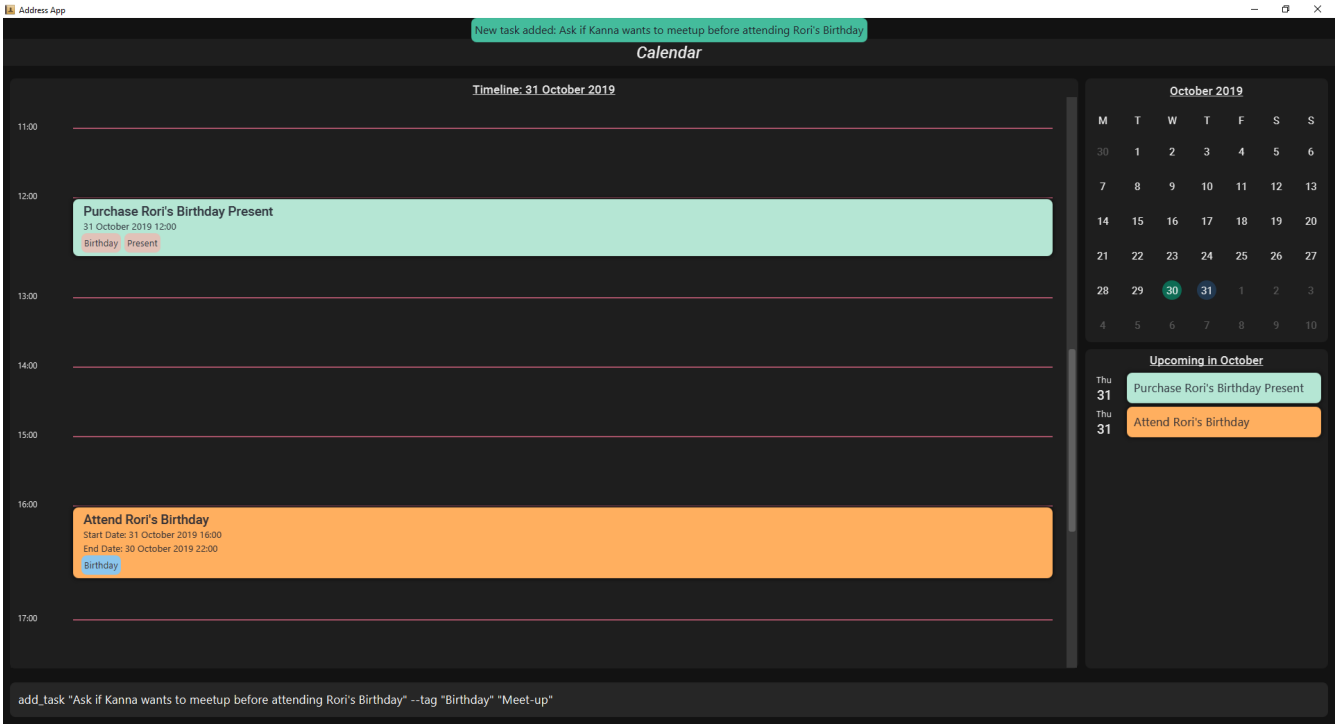


Figure 21. Current design of the UI

Aspect: Design of the `CalendarPanel`

- Alternative 1: The `CalendarPanel` is of an actual calendar, depicting a number events and tasks on each day of the month.
    - **Pros**: It will provide a better representation of a calendar, allowing people to judge how much is going on in a day of that month in one look.
    - **Cons**: Due to the nature of how limited in size a calendar can be, the user will be required to either check `ListPanel` for the details of an event or tasks, or have an extra screen beside the calendar for the user to check the details.
    - **Cons**: Similarly, a calendar can only input up to a fix amount of events or tasks there are on a particular day.

- Alternative 2 (current choice): The `CalendarPanel` consists of a mini-calendar as well as a timeline. An additional slot for upcoming events and tasks was later designed with the increase in space.
    - **Pros**: There is a much more greater space to show how much events or tasks one can have in a day, week or month.
    - **Pros**: The user can easily managed and check the Events and Tasks of a certain day.
    - **Cons**: Even though it is a timeline, it is still rather similar to list view, just with the timeline added to limit the amount of events seen on that day, week or month.
    - **Cons**: :The user will not be able to know what Events or Tasks there are easily, unless they change the view to month view. However, the increase space allows a small section for the upcoming events and tasks which tackles this problem.

Aspect: Design of the `LogPanel`

- Alternative 1: The `LogPanel` is placed side-by-side with any other panel.
    - **Pros**: The users can always have a visualization of the success of their commands
    - **Cons**: A large portion of the space is used for the `LogPane`, even if the it is scaled down compared to the other panels.
    - **Cons**: Appearance-wise, it looks extremely clunky due to most of the users' time will be looking at the calendar or list itself instead of the log.
- Alternative 2 (current choice): The `LogPanel` is placed separately as a different panel which can be access at any time from other panels. After each command is typed, a pop-up box will appear to indicate the success or failure of the command.
    - **Pros**: Since the user most of the time would only want to know if their command is successful or not, the pop-up box will be sufficient for such indication.
    - **Cons**: The user will have to check the `LogPanel`

The initial design is as of the image above showing the old UI. However, we decided to scrape it and did an overhaul of the UI using alternative 2 instead. This is due to our decision of wanting a better-looking and minimalist UI instead of one packed with information.

5. Documentation

Refer to the guide here.

6. Testing

Refer to the guide here.

7. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile:

- is a student
- has a need to manage their Events and Tasks for visualization.
- requires reminders for their Events and Tasks.
- prefer desktop apps over other types
- can type fast
- prefers typing over mouse input
- is reasonably comfortable using CLI apps

Value proposition: manage Reminders as well as viewing Events and Tasks much faster than a typical mouse/GUI driven app

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a …	I want to …	So that I can…
* * *	new user	see usage instructions	refer to instructions when I forget how to use the App
* * *	user	add an Event or Task	keep track of an Event or Task that I have in the future
* * *	user	delete an Event or Task	remove the Event or Task I no longer need.
* * *	user	find an Event or Task by name	locate the details of the Event or Task without having to go through the entire list
* * *	user	find an Event or Task by tags	remember the details of the Event or Task that I forget about

Priority	As a …	I want to …	So that I can…
* * *	user	undo and redo commands	undo any commands which wrongly inputted
* * *	user	edit my Events and Tasks	change the details of the event, be it location, date or time
* * *	user that works on multiple computer	port my data between computers	keep track on all my computers.
* * *	student	have constant reminders to track the deadline of my assignments	not forget to complete and submit them
* * *	student	keep track of how long it takes for me to complete a task	gauge how long I will need to take for future similar tasks
* * *	student with weekly assignments and lectures	have my reminders to be recurring	be reminded without having to input the information in again
* * *	busy student	have a convenient way to visually see my assignments and projects	complete them in the right priority
* * *	busy user	be informed if any different events clash with each other	understand which event to prioritize or reschedule

Priority	As a …	I want to …	So that I can…
* *	user	add a contacts	add them into Events to remind myself who I am meeting up with
* *	user	archive my completed Tasks	remind myself if I complete a task but forgot about it
* *	user	create custom commands that contain the execution of multiple sub-commands	quickly input in a command without the need to edit it
* *	student	visualize my timetable	plan for when it is time to take a break from studying
* *	student	find a time for my project teammates to meet up	schedule a meeting without clashing together with other events
*	user	import contacts in vCard format	integrate them with my events
*	user	export contacts in vCard format	integrate them with my other computers
*	student	keep track of sub-tasks in a main task	know my current progress in a report

{More to be added}

Appendix C: Use Cases

(For all use cases below, the **System** is the `Horo` and the **Actor** is the `user`, unless specified otherwise)

Use case 1: Add a Task

MSS

1. User requests to add a Task
2. Horo replies that the Task has been added

   Use case ends.

Extensions

- 1a. The user adds additional sub-commands to the Task command

  Use case ends.

- 2a. The given add Task command is of the wrong format.

    2a1. Horo displays an error message.

    Use case resumes at step 1.

Use case 2: Delete a Task

MSS

1. User requests to delete a specific Task from the already displayed list
2. Horo deletes the Task

   Use case ends.

Extensions

2a. The given delete Task command is of the wrong format.

    2a1. Horo displays an error message.

    Use case resumes at step 1.

Use case 3: Find a Task by name

MSS

1. User requests to find a Task
2. Horo displays the list of Task with the keywords found in its name

   Use case ends.

Extensions

2a. The given find Task command is of the wrong format.

2a1. Horo displays an error message.

Use case resumes at step 1.

Use case 4: Undo and Redo commands

MSS

1. User requests to add an Task
2. Horo replies that the Task has been added
3. User requests to undo the command
4. Horo replies that the previous command has been undone

Use case ends.

Extensions

- 1a. The user adds additional sub-commands to the Task command

  Use case ends.

- 2a. The given add Task command is of the wrong format.

  2a1. Horo displays an error message. Use case resumes at step 1

- 4a. User decides the to Redo the added Task

  3a1. Horo replies that the added Task has been redone

  Use case ends

Use case 5: Edit a Task

MSS

1. User requests to add a Task
2. Horo replies that the Task has been added
3. User request to edit a Task with the sub-commands
4. Horo replies that the Task has been edited

Use case ends.

Extensions

- 1a. The user adds additional sub-commands to the Task command

  Use case ends.

- 2a. The given add Task command is of the wrong format.

    2a1. Horo displays an error message.

    Use case resumes at step 1.

- 4a. The given edit Task command is of the wrong format.

    4a1. Horo displays an error message.

    Use case resumes at step 3.

    {More to be added}

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 Events and Tasks without a noticeable sluggishness in performance for typical usage.
3. Should function on both `32-bit environment` and `64-bit environment`
4. Should work without any internet required.
5. A user with above average typing speed for regular English text (i.e. not code, not system admin commands) should be able to accomplish most of the tasks faster using commands than using the mouse. .

{More to be added}

Appendix E: Glossary

Mainstream OS

Windows, Linux, Unix, OS-X

Event

A thing that happens or takes place during a certain period of time, or of a general time.

Task

A piece of work that is to be completed or taken note of.

Appendix F: Product Survey

reminder-bot on Discord

Author: JellyWX

Pros:

- A reminder bot on a popular voice and text chat application

- Capable of parsing english language as compared to CLI styled commands

Cons:

- Lack of visualization of the Events and Tasks
- Parsing english language makes it slower to type with a longer requirement as compared to CLI styled commands

Google Calendar

Company: Google

Pros:

- A Calendar application that is capable of storing Events and Tasks as well.
- Mostly uses GUI for interaction with user instead of having CLI, favouring to the common crowd.

Cons:

- Mostly uses GUI for interaction with user instead of having CLI, which does not favour those who prefers CLI.
- It requires an account to be usable.
- The desktop version requires a browser, which in turn requires Internet and hence not offline.

Appendix G: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE	These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

G.1. Launch and Shutdown

1. Initial launch
    a. Download the jar file and copy into an empty folder
    b. Double-click the jar file
    Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.
2. Saving window preferences
    a. Resize the window to an optimum size. Move the window to a different location. Close the window.
    b. Re-launch the app by double-clicking the jar file.
    Expected: The most recent window size and location is retained.

{ more test cases … }